A RAND NOTE

Extracting Tactical Data from Operation Orders

James R. Kipps, Jed B. Marti

RAND

The research described in this report was sponsored by the United States Army under Contract No. MDA903-91-C-0006.

Published 1992 by RAND
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90407-2138

A RAND NOTE

N-3300-A

Extracting Tactical Data from Operation Orders

James R. Kipps, Jed B. Marti

Prepared for the
United States Army

RAND

PREFACE

This Note describes interim results of the User-Assisted Translation of Operational Plans (UATOP) project. It is of interest to those wishing a technical overview of a new machine-translation approach to automating the extraction of important data from operation orders (OPORDs). It is also of interest to those wishing to use, understand, and maintain a translation system, called OPORG, which translates OPORDs on UNIX and MS-DOS computers. This work is part of a larger effort to develop a tool for anticipating combat ammunition consumption through advanced simulation techniques. The OPORG system described herein is a precursor to another translation system, called OPSCRIPT, for extracting "Task to Maneuver Unit" data from given OPORDs.

Our approach to OPORD translation is of general interest because it takes advantage of the highly organized structure of OPORDs. The same technology being developed as part of the UATOP effort is also applicable for extracting data from a wide range of other highly structured but "not machine-readable" documents.

This Note is written for a technical audience. Familiarity with concepts in compiler construction and formal language theory is assumed on the part of the reader. A useful textbook is *Principles of Compiler Design* (Aho and Ullman, 1977). Familiarity with the RAND Compiler Kit (RACK) is also assumed. A general discussion of RACK appears in "RACK: A Parser Generator for AI Languages" (Kipps, 1990). The RAND Note, *The RAND Compiler Kit (RACK): Reference Manual and User's Guide* (Kipps, 1991), provides details of the system's operation.

This work has been sponsored by the Combined Arms Support Command, Ft. Lee, Virginia, under the direction of LTG Leon Salomon. This work has been conducted as part of the Force Employment Program of the Arroyo Center. Questions involving technical issues should be addressed to Dr. Jed Marti, Project Leader.

THE ARROYO CENTER

The Arroyo Center is the U.S. Army's federally funded research and development center (FFRDC) for studies and analysis operated by RAND. The Arroyo Center provides the Army with objective, independent analytic research on major policy and organizational concerns, emphasizing mid- and long-term problems. Its research is carried out in four programs: Strategy and Doctrine; Force Development and Technology; Military Logistics; and Manpower and Training.

Army Regulation 5-21 contains basic policy for the conduct of the Arroyo Center. The Army provides continuing guidance and oversight through the Arroyo Center Policy Committee (ACPC), which is co-chaired by the Vice Chief of Staff and by the Assistant Secretary for Research, Development, and Acquisition. Arroyo Center work is performed under contract MDA903-91-C-0006.

The Arroyo Center is housed in RAND's Army Research Division. RAND is a private, nonprofit institution that conducts analytic research on a wide range of public policy matters affecting the nation's security and welfare.

Lynn E. Davis is Vice President for the Army Research Division and Director of the Arroyo Center. Those interested in further information about the Arroyo Center should contact her office directly:

Lynn E. Davis
RAND
1700 Main Street
P.O. Box 2138
Santa Monica, CA 90407-2138

SUMMARY

The User-Assisted Translation of Operational Plans (UATOP) project and Anticipating Combat Ammunition Consumption (ACAC) project are building tools to assist Division Ammunition Officers (DAOs) to anticipate ammunition consumption before battle. This capability is a vital component of logistics operations in the Army's emerging AirLand Operations doctrine. Operating with future Army information systems, the program will examine operation orders (OPORDs) and simulate the battle and potential courses of action to estimate consumption quantity and location of high-cost, high-weight munitions. Significantly improving the DAO's capability will increase combat unit effectiveness by minimizing the amount of extra ammunition that must be carried by a unit if oversupplied, thus freeing up vehicles and enabling more rapid deployment. Likewise, the system will pinpoint potential undersupply problems before the battle.

A key to the ACAC simulation technology is the ability for the simulation to "understand" the contents of OPORDs. This includes establishing task force organization and locating and translating the tasks to perform. Because OPORDs contain English directives written for people and not machines, current machine-translation techniques are not directly suitable for extracting OPORD data. We have developed a new machine-translation approach that takes advantage of the prescribed five-paragraph format of OPORDs to identify and isolate pertinent pieces of information. This approach uses concise and clear rules to automatically generate programs that take as input textual OPORDs such as those transmitted through the Maneuver Control System (MCS), extracts the desired data, and outputs this data as input to other computer systems.

In this Note, we describe our approach to automating the extraction of OPORD data. We describe an application of our approach to the task of generating the OPORG translation system, which extracts task organization data from input OPORDs. OPORG itself is not a significant system; rather, it is an interim product used to demonstrate our technique for OPORD data extraction. A larger system, OPSCRIPT, has also been implemented using this approach. OPSCRIPT extracts tactical data from division-level OPORDs and outputs simulation scenarios for use by the ACAC model. The OPSCRIPT system is not

described here but will be described in a future document. While the focus of the UATOP project is on extracting data from OPORDs, the techniques described in this Note are generally applicable to extracting and checking data from a wide range of highly structured but not "machine-readable" documents.

ACKNOWLEDGMENTS

The authors would like to thank LTC James Price for many helpful discussions of the problems addressed in this Note and Brian Leverich for improving its presentation.

CONTENTS

FIGURES

1. INTRODUCTION

BACKGROUND

Combat orders set forth the details of tactical operations and field administration. In particular, an operation order (OPORD) is *a directive issued by a commander to subordinate commanders for the purpose of effecting the coordinated execution of an operation* (AFSC, 1984). OPORDs deal with operation specifics and dictate the conduct of tactical operations and movements. Military software systems for battle management, logistics, war-gaming, and combat training, such as JANUS (U.S. Army, unpublished documentation) and SCALP (U.S. Army, 1989), as well as other DoD automation projects currently under development, require the tactical data contained in OPORDs.

Although OPORDs are transmitted through the Maneuver Control System (MCS), they cannot be directly input to military software systems. This data must be extracted by slow, costly, and error-prone manual effort. The problem stems from the fact that OPORDs are not written for machines but for people. Conventional machine-translation technology expects input text to conform to a formal syntax, such as that of a computer programming language; grammar and typing errors must be repaired before text, such as computer programs, can be translated properly. While OPORDS are highly structured documents, they are not computer programs. OPORDs largely consist of unconstrained and (occasionally) ungrammatical English directives. They often contain spelling and typing errors, many of which do *not* need to be corrected for an OPORD to be utilized by human readers. As a result, OPORDs cannot be automatically translated with conventional technology.

OBJECTIVE

The User-Assisted Translation of Operational Plans (UATOP) project has sought to overcome the problems associated with the automatic extraction of tactical data from OPORDs. The overall objective has been to develop a partially automated approach to the translation of division-level OPORDs into simulation scenarios for use by the Anticipating Combat Ammunition Consumption (ACAC) model. This objective has been reached and a prototype translation system (OPSCRIPT) has been implemented. A smaller system, called OPORG, which

runs in a personal computer (PC) environment, has also been implemented using this approach. OPORG extracts task organization data from an input OPORD and outputs this data in a format that makes it readily accessible by other software systems.

APPROACH

Our approach to OPORD translation uses a hybrid of conventional machine-translation technology. It takes advantage of the highly organized structure of OPORDs in order to identify individual pieces of interest, to which other technologies, such as natural language parsing, can be applied. It should also be noted that our approach is not necessarily restricted to OPORDs. The same technology being developed as part of the UATOP effort is generally applicable for extracting data from a wide range of highly structured but "not machine-readable" documents and might also be useful for detecting errors in such documents. While this work is being conducted in support of ammunitions logistics, management of other classes of supply, such as spare parts, major end items, and fuel, requires a similar ability to extract data from OPORDs.

ORGANIZATION OF THIS NOTE

In this Note, we describe our approach to OPORD translation as it applies to the OPORG system. Although the OPORG system is not significant in itself, it does provide an illustrative example for explaining our approach and for understanding the substantially larger, and more significant, OPSCRIPT system. We will describe the OPSCRIPT system in a subsequent document. In Section 2, we define the extraction problem for the OPORG system. In Section 3, we outline our approach to the extraction problem and overview the system architecture implementing this approach. The next three sections are intended for a technical audience with exposure to concepts from machine translation and parsing theory; introductions are provided to allow nontechnical readers to understand the flavor, if not the details, of the material. In Sections 4 and 5, we describe the two major components of the system architecture, namely, the document mapper generator (DMG) and the RAND Compiler Kit (RACK); we describe how the DMG and RACK are used to create the OPORG translator system in Section 6. We make our closing remarks in Section 7.

2. DEFINING THE OPORG EXTRACTION PROBLEM

The OPORG translation system addresses the problem of extracting task organization data from an input OPORD and making that data accessible to other software systems. As we explained in Section 1, the difficulty in automating the extraction of OPORD data is the lack of a formal syntax for OPORDs and the fact that OPORDs consist primarily of English text. The OPORG system must be able to recognize where pertinent pieces of data begin and end in the text, which requires matching complex character patterns and keeping track of which patterns have been seen and which come next. Manually writing and debugging a program to do this task could take up to six man-months of effort. Maintaining such a program, or modifying it to extract additional data, would be considerably and, most likely, prohibitively expensive.

There are, however, several positive aspects to OPORDs with regard to automatic machine translation. OPORDs *are* highly structured documents that follow a prescribed format. While this is not the same as a formal syntax, it is similar. In addition, the major and minor elements of OPORDs are introduced by distinctive character patterns, and conventional machine-translation technology does provide techniques for automatically generating programs for matching character patterns. With regard to the "natural language" aspect of the problem, OPORDs are expected to reflect a commander's intention. Essential characteristics of an OPORD include clarity, brevity, simplicity, completeness, and authoritative expression. Indecisive, vague, and ambiguous language leads to uncertainty and, thus, does not characterize well-written OPORDs. Consequently, the English directives within OPORDs are constrained in a way that should aid machine translation. This characteristic of OPORDs is taken advantage of in the OPSCRIPT translation system but is not pertinent to the OPORG system.

OPORD FORMAT

We assume that the input OPORD is accessible as an on-line ASCII text file and that it is written in standard, five-paragraph format, as prescribed in Field

Manual (FM) 100-5 and AFSC (1984). The general organization of five-paragraph format is outlined below.

1. Heading
   (a) Issuing Unit
   (b) DTG Effective
   (c) Operation Order Number
   (d) References
   (e) Task Organization
2. Body
   (a) Situation
   (b) Mission
   (c) Execution
   (d) Service Support
   (e) Command and Signal
3. Ending
   (a) Authentication
   (b) Annexes
   (c) Distribution

Sections, paragraphs, subparagraphs, and items within an OPORD are enumerated, labeled, or otherwise introduced by identifiable lexical cues. These lexical cues aid human readers in recognizing and extracting pertinent information from an OPORD and are key to our automated extraction of OPORD information.

WHAT OPORG EXTRACTS

The OPORG translation system extracts task organization data from a given OPORD and outputs this data in a form that is generally accessible by other software systems. In particular, OPORG is responsible for extracting the following data.

1. Operation order number, issuing unit, and date-time group (DTG) the OPORD is effective.
2. Task organization data for each subordinate unit in the task organization section. This includes the unit designation, unit location, unit attachment, and DTG the attachment is effective.
3. DTG the operation is to begin.

WHAT OPORG OUTPUTS

To make the task organization data extracted by OPORG generally accessible to other software systems, we selected an output format that groups related data on distinct lines of output. Data fields within each line are separated by easily recognizable delimiter characters.

Four record types are created by OPORG. The first character of each line of output identifies its record type. The following record types are created:

0/*opord no*/*issuing unit*/*DTG order effective*//

This record contains the operation order number, the issuing unit designation, and the DTG the OPORD is effective. This record will appear only once in the output file.

1/*unit des*/*unit loc*/*attached to*/*DTG effective*//

This record contains data from the task organization section of the OPORD, namely, a unit designation, unit location, the unit to which it is being attached, and the DTG effective (if given). The record will appear once for each unit or task force in the OPORD.

2/*comment*//

This record contains a comment that should be ignored when reading the OPORG-generated file; comments are used to highlight portions of OPORD text from which data is extracted for the purpose of manual verification.

3/*DTG operation to begin*//

This record contains the DTG for the beginning of the operation. Data fields are separated by a slash (/) and data lines are terminated by a double slash (//).

As an example, consider the OPORD segment appearing in Fig. 1. The pertinent lines from which OPORG extracts its data are underlined. The output generated by OPORG for this example appears in Fig. 2.

The first output record gives the OPORD number (49003), the issuing unit (699TH Mechanized Infantry Division), and the DTG the OPORD is effective (261200U0590). This record is followed by three task force records. Each task force record is followed by a comment record, which highlights the text from which

```
UNCLASS
EXER/ROTATION 90-10/BLUE//
MSGID/ORDER/699TH INF DIV (M)/49003/JUN//
REF/A/OPLAN 90-10/699TH INF DIV (M)/251200 MAY 90//
AMPN/REF/A IS 699TH INF DIV (M) BASIC ORDER//
ORDTYP/FRAGORD/699TH INF DIV (M) 90-10-1//
MAP/ V795/ FORT GOLGIFRANCHIA/MIM NORTH/2-DMA//
MAP/1501/NI  11-1,11-2,11-3,11-5,11-6//
TIME ZONE/U//
ORDREF/OPLAN 90-10/699TH INF DIV (M)//

HEADING/TASK ORGANIZATION//
5UNIT
/UNITDES      /UNITLOC /CMNTS
/TF 2-13 MECH /NL1702  /OPCON TO 699TH AB AS DIV TCF EFFECTIVE
                       021200 JUN 90.
/TF 1-61 AR   /NL6392  /OPCON TO 3D BDE (LIVE FIRE) EFFECTIVE
                       ON CLOSURE IN 3D BDE M.
/TF 3-4 AVN   /AREA B  /OPCON TO 1ST BDE EFFECTIVE 291300 MAY 90//

GENTEXT/SITUATION/
A. ENEMY: SEE CURRENT INTSUM TO ANNEX B (INTELLIGENCE).
B. FRIENDLY:
   (1) (A) 10TH (US) CORPS:  DEFEND IN SECTOR ALONG PL OLYMPIA
NLT 020001 JUN 90 TO DEFEAT ATTACKS OF THE 16TH (KRAS) CAA TO
PROTECT THE USJTF LOGISTICAL BUILD-UP AND THE DEPLOYMENT OF THE
20TH (US) CORPS FOR THE ASSUMPTION OF OFFENSIVE OPERATIONS TO
RESTORE THE GOLGIFRANCHIA-LILLIPUTIA INTERNATIONAL BORDER.
       (B) CORPS CONCEPT:  THE INTENT OF THIS OPERATION IS TO
RETAIN TERRAIN KEY TO FUTURE OFFENSIVE OPERATIONS AND TO GAIN
TIME FOR THE DEPLOYMENT OF THE 20TH (US) CORPS INTO GOLGIFRANCHIA.
   (2) 698TH INFANTRY DIVISION (M): DEFEND IN SECTOR NLT 020001
JUN 90 TO CONTAIN ENEMY FORCES WEST OF THE CADY MOUNTAINS.
   (3) 697TH ARMORED DIVISION: CORPS RESERVE: BE PREPARED TO
COUNTERATTACK TO DESTROY REGIMENTAL SIZED PENETRATIONS OR LARGER
OF FORWARD DIVISION REAR BOUNDARIES (PL PALOUSE).
   (4) 696TH ACR: CONDUCT DEFENSIVE COVER NLT 020001 JUN 90 TO
PREVENT BYPASS OR ENVELOPMENT OF THE CORPS RIGHT (NORTH) FLANK.
MAINTAIN CONTACT WITH 21 ST (US) ABN CORPS.
   (5) 695TH AB: ATTACK, ON ORDER, TO ATTRIT AND DELAY THE
2D ECHELON REGIMENTS OF ATTACKING MRD FORWARD OF 698TH ID (M).
   (6) ELEMENTS OF THE 695TH TAF SUPPORT THE 10TH (US) CORPS//

GENTEXT/MISSION.699TH INF DIV (M):  DEFENDS 020001 JUN 90 ALONG PL
OLYMPIA TO CONTAIN ENEMY FORCES WEST OF THE BANDOLIER (NL6071), CRACKER
(NL6993) AND DACHANEE (NL4316) MOUNTAIN PASSES TO PROTECT THE USJTF
LOGISTICAL BUILD-UP AND THE DEPLOYMENT OF THE 20TH (US) CORPS//
```

Fig. 1—Segment of Text from Operation Order

```
0/49003/699TH INF DIV (M)/251200U0590//
1/TF 2-13 MECH/NL1702/699TH AB AS DIV TCF/021200U0690//
2/OPCON TO 699TH AB AS DIV TCF EFFECTIVE 021200 JUN 90//
1/TF 1-61 AR/NL6392/3D BDE (LIVE FIRE)///
2/OPCON TO 3D BDE (LIVE FIRE) EFFECTIVE ON CLOSURE IN 3D BDE M//
1/TF 3-4 AVN/AREA B/1ST BDE/291300U0590//
2/OPCON TO 1ST BDE EFFECTIVE 291300 MAY 90//
3/020001U0690//
```

Fig. 2—Sample OPORG Output

the unit's attachment data was extracted. Because TF 1-61 AR is attached to 3D BDE (LIVE FIRE) effective on closure in 3D BDE AA, instead of at a particular DTG, the DTG attachment effective field for this entry is empty.

3. OVERVIEW OF THE OPORG SYSTEM ARCHITECTURE

In our approach to extracting OPORD data, we factor the translation task into two phases. In the first phase, *lexical cues* (character patterns matching such things as section headings and item numbers) are used to recognize the major and minor elements of an OPORD, such as sections, subsections, and items; the text contained in the body of these elements is ignored during this phase.

In the second phase, pertinent elements of the OPORD are identified, and their associated text is passed to special-purpose translators (or *parsers*). Each parser identifies the syntactic structure of the input text according to a given grammar. By identifying its syntactic structure, a parser can map the corresponding text into useful data structures.

Finally, the parsed results are then collated, and the extracted data output. For the OPORG extraction task (as defined in Section 2), the special-purpose parser recognizes DTGs; for the OPSCRIPT system, there is an *ad hoc* natural language parser that outputs scripts for the ACAC simulation module. Although this approach does not yet address all issues in OPORD translation, such as error handling, it does provide a mechanism for focusing on the elements of interest within an OPORD and for dealing with those elements individually.

MAJOR SYSTEM COMPONENTS

The OPORG system takes as input an OPORD text file, extracts task organization plus other data, and outputs this data in a task organization file, the format of which is described in Section 2. This system architecture and data flow are shown in Fig. 3. There are three major components.

- **OPORD Mapper.** The OPORD Mapper (or *mapper*) takes as input the OPORD text file and outputs a hierarchical data structure called a *file map.* The file map identifies the major and minor elements of the OPORD text file.
- **Data Extractor.** The Data Extractor (or *extractor*) traverses the file map generated by the mapper from the bottom up, looking for elements with associated *extractor actions.* The extractor executes these actions and passes the results up the file map as arguments to higher-level extractor actions. The lowest-level extractor actions typically apply

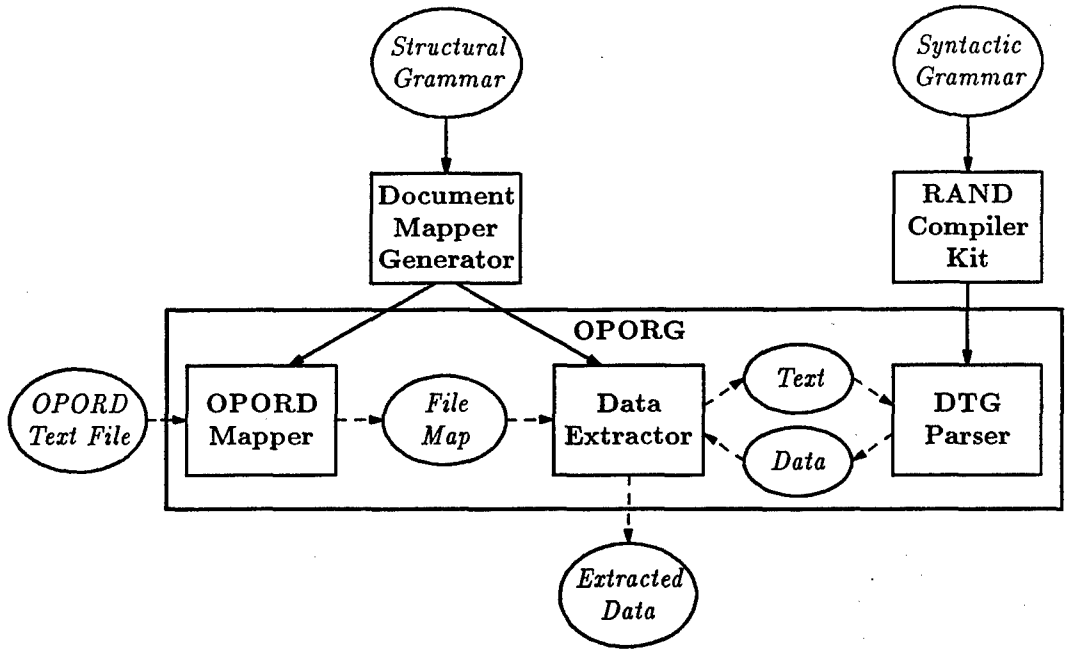

Fig. 3—OPORG System Architecture

special-purpose parsers to the text of their associated element, while the highest-level actions typically output the extracted data.

- **DTG Parser.** The DTG Parser (or *parser*) is the primary parser whose purpose is to recognize and extract a DTG from a segment of text. The data structure representing the DTG is then returned to the Data Extractor.

As we explain next, the mapper, extractor, and parser components of OPORG are generated automatically from grammars. This is depicted in Fig. 3 with the use of solid arrows. Data flow through the OPORG system is depicted with dashed arrows.

USING A "GENERATOR" APPROACH WITH OPORG

Experience with computer programming languages has taught language developers that translation systems, even for simple languages, are nontrivial to construct by hand. For instance, the first Fortran compiler required 10 man-years of effort. Since then, a number of software tools, variously called compiler-compilers, parser-generators, and translator-writing systems, have been developed specifically to help construct compilers, parsers, and other forms of translators. YACC (Johnson, 1975) is an example of a commonly used parser

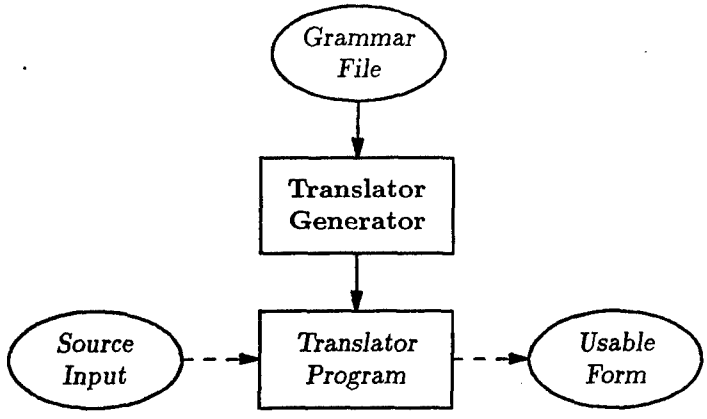

Fig. 4—Generic Translator Generator

generator for languages developed under UNIX.[1] As depicted in Fig. 4, a translator generator is a computing system that outputs a program for translating source input in some *target language* into a usable form. The input to the translator generator is a *grammar file* describing the syntax of the target language. Again, generation is depicted with solid arrows, and data flow through the translator is depicted with dashed arrows.

We apply a similar "generator" approach to constructing the mapper, extractor, and parser components of an OPORD translation system such as OPORG. The OPORD Mapper and the Data Extractor are generated by a system called the Document Mapper Generator (DMG), while the special-purpose parser is generated by the RAND Compiler Kit (RACK). The input to the DMG is a grammar file that describes the *hierarchical structure* of a document, such as an OPORD, and the lexical cues that identify particular elements of the document. The grammar file also specifies extractor actions to be executed when particular structure elements are recognized. The input to RACK is a grammar file that describes the *syntactic structure* of particular text elements. RACK was developed at RAND for generating parsers for advanced programming languages (Kipps, 1990). The DMG was developed as part of the UATOP project specifically for extracting OPORD data. The DMG can be applied to other types of highly structured, but not machine-readable, documents.

[1] UNIX is a trademark of Bell Laboratories.

4. MAPPING DOCUMENT STRUCTURE WITH DMG

The Document Mapper Generator (DMG) constructs mapper and extractor programs for highly structured documents. The mapper program is responsible for identifying the major and minor structural elements of an input document, and the extractor is responsible for calling the parser, for collating the data returned by the parser, and for writing the output.

The input to the DMG is a grammar file that describes the hierarchical structure of target input documents, such as OPORDs. The output is a mapper file called `y.map.c`, which contains the source code for both the mapper and the extractor. The mapper and extractor routines are implemented in the C programming language. The extractor routines automatically interface with the C-based parsers generated by RACK.

The grammar file primarily consists of a set of *structure rules* that identify the pertinent structural elements of the input documents and their subcomponents. Structure rules have the general appearance of rules from a context-free grammar. There are two major differences. First, structure rules can be defined iteratively but not recursively. This is done in recognition of the fact that documents such as OPORDs have a structure consisting largely of ordered sections and subsections, the purpose, levels, and structure of which are known a priori. Second, structure rules typically contain one or more *regular expressions* (character patterns) that recognize lexical cues. The DMG depends on the existence of lexical cues to introduce and delimit elements of interest in the document. Lexical cues consist of such things as section and subsection headings and item numbers; they indicate the introduction or ending of a structural element.

In this section, we provide a technical discussion on the operation of the DMG as it applies to the OPORG translation system. The reader is assumed to be familiar with the RACK translator-generator system (Kipps, 1991) and its syntax. The reader is also assumed to be familiar with concepts in compiler construction, LR grammars, and regular expressions, a discussion of which can be found in Aho and Ullman (1972; 1977).

DMG GRAMMAR FILES

A DMG grammar file has the following basic format:

Declarations
%%
Structure Rules
%%
Code.

The *declarations* section is used to insert user code into the extractor program, to define lexical macros, and to specify the root of the grammar. The following declarations are recognized.

{ *code* }

For specifying code segments. All characters enclosed in curly braces are placed at the front of the mapper file.

%start *name*

Defines a symbol *name* as the root structure element of the grammar.

%def *macro* (*re*) [;]

Assigns the regular expression *re* to the lexical macro *macro*, a symbol or nonalphabet character prefixed by a percent sign (%). When *macro* is encountered in the body of a structure rule, it is replaced by *re*.

%noextractor

Tells the DMG to disable the extractor code in the mapper file. This is needed when interfacing to LISP, which is done with the OPSCRIPT system; OPSCRIPT is not described in this Note.

The *structure rules* section consists of one or more *structure rules*, which describe the hierarchical structure of input documents. The *code* section is used to define additional subroutines, such as `main()`, to be added to the extractor program. Each section is separated by double percent signs (%%).

STRUCTURE RULES

Structure rules have the general form

lhs : *rhs* [*action*]

where the *lhs* is a symbol naming a *structure element* of the document, the *rhs* describes the structure of that element, and the optional *action* specifies C code to be executed by the extractor program when the *lhs* element is recognized in an input document. Each element in a structure's *rhs* can be separated by an

arbitrary number of white space characters: blanks, tabs, and newlines. The *rhs* of a rule can consist of an arbitrary sequence of the following items.

Regular expressions

Regular expressions are character patterns used to describe *lexical cues*. Regular expressions are delimited by parentheses; their syntax is identical to that used by RACK, as defined in Kipps (1991). Regular expressions and their use in OPORG are described in detail later in this section.

Random text elements

Random text elements are denoted by empty angle brackets (<>) and are used to describe portions of the document to be ignored by the mapper and extractor programs.

Nonterminal elements

Nonterminal elements are denoted by a symbol delimited by angle brackets (<*nonterminal*>). These are passed to the extractor program as they identify portions of text that must be parsed.

Structure elements

Structure elements are symbols identifying other structure rules. They are used to describe hierarchical document structure.

Optional elements

Optional structure elements are delimited by brackets ([*element*]).

Alternate elements

Two or more symbols delimited by square brackets and separated by bars ([$structure_1|\cdots|structure_n$]) denote disjunction over the named elements.

Repeated elements

A structure element immediately followed by a star (*) can appear zero or more times in a document, and if followed by a plus (+), it can appear one or more times.

Regular expressions, random text, and nonterminals denote basic "leaf" elements of a document; structure elements denote compound elements. Structure elements may not be defined recursively.

THE OPORD STRUCTURE GRAMMAR

In the remainder of this section, we will describe the meaning and use of various pieces of the OPORD structure grammar used in the OPORG system, which appears in Fig. 5. We will describe constructs of importance as they appear in the grammar. The first panel of Fig. 5 contains the declarations and structure rules.

The Declarations Section

The declarations section of the grammar contains three items. The first declaration

```
{
#include <stdlib.h>
#include "ytab.h"
struct dtgroup *opdtg;
}
```

is a segment of user code; the characters between the curly braces are inserted directly at the top of the mapper file. The file `stdlib.h` is a standard C library file; `ytab.h` is generated by RACK and contains declarations that enable the extractor to interface with the parser. The effective DTG (date-time group) of the operation will be assigned to variable opdtg by an extractor action; the structure dtgroup is defined in the grammar file for the DTG parser, which is described in Section 5.

The next declaration

```
%def %~ ([ \t]*)
```

defines `%~` as a lexical macro (see Section 5); all instances of `%~` occurring in regular expressions will be replaced by the expression `[ \t]*`, which matches zero or more blanks or tabs. The last declaration

```
%start opord
```

specifies that opord is the root structure element.

The Structure Rules

The grammar also contains 14 structure rules, one for each of 14 structure elements. The first structure rule

```
opord: heading body ending
```

defines the root element opord as consisting of three parts: a heading, followed by a body, followed by an ending.

```
{
#include <stdlib.h>
#include "ytab.h"
struct dtgroup *opdtg;
}
%def %~ ([ \t]*)
%start opord
%%
opord:      heading body ending
heading:    <> msg_id <> msg_ref_no <> zone_used <> task_org
body:       <> mission <>
ending:     <>
msg_id:     "^MSGID%~/%~ORDER%~/<>/<>/<>//"
            { printf("0/"); prtmap($4); printf("/"); prtmap($2); }
msg_ref_no: "^REF%~/%~{[A-Z]}%~/<>/<date_time_group>//"
            { opdtg = $4->value.dtg; }
zone_used:  "^TIME ZONE<time_zone>//"
            { printf("/"); prdtg(opdtg); printf("//\n"); }
task_org:   "^HEADING%~/%~TASK ORGANIZATION//"
            <>
            "^/UNITDES /%~UNITLOC /%~CMNTS"
            task_entry+
task_entry: unitdes unitloc cmnts
unitdes:    (\n? '/'?) (%P+ (' ' %P+)*) ([ \t]+)
            { printf("1/"); prtmap($2); }
unitloc:    ('/'?) (%P+ (' ' %P+)*) ([ \t]+)
            { printf("/"); prtmap($2); }
cmnts:      ('/'?) cmnttxt ('//'|'.')
            { printf("2/"); prtmap($2); printf("//\n"); }
cmnttxt:    "{(OPC|ATT)[A-Z]*} TO <> EF<date_time_group>"
            { printf("/"); prtmap($2);
              printf("/"); prdtg($4->value.dtg);
              printf("//\n"); }
mission:    "^GENTEXT%~/%~MISSION.<>:<date_time_group>//"
            { printf("3/"); prdtg($4->value.dtg); printf("//\n"); }
```

Fig. 5—Example OPORD Structure Grammar (Rules)

```
%%

prtmap(tmap)
  ZZTMAP *tmap;
  { char *ptr;
    zzfilltmap(tmap);
    for(ptr = tmap->nxptr; ptr < tmap->ndptr; ++ptr)
      {
        if(isspace(*ptr))
          {
            putc(' ', stdout);
            while(isspace(*++ptr) && ptr < tmap->ndptr) ;
            --ptr;
          }
        else
          putc(*ptr, stdout);
      }
  }

main(argc, argv)
  int  argc;
  char *argv[];
  { ZZTMAP *fmap;
    fmap = zzmap(argc, argv);
    /* DEBUG: zzwrttmap() outputs entire tmap hierarchy in human
       readable form.
    if(fmap)
      zzwrttmap(fmap, stderr);
    else
      fprintf(stderr, "failure\n");
    */
    if(fmap)
      zzextract(fmap);
  }
```

Fig. 5 (cont.)—Example OPORD Structure Grammar (Code)

The second rule

```
heading: <> msg_id <> msg_ref_no <> zone_used <> task_org
```

defines the structure of heading in terms of the pertinent pieces of an OPORD's heading; the other portions of the heading are treated as random text (<>).

Likewise, the third rule

```
body: <> mission <>
```

defines the structure of body in terms of a single element, mission, ignoring everything before and after. Because we are not interested in anything else, ending is defined as consisting entirely of random text.

The rule

```
msg_id: "^MSGID%^/%^ORDER%^/<>/<>/<>//"
   { printf("O/"); prtmap($4); printf("/"); prtmap($2); }
```

defines the structure of an OPORD's message identifier, from which OPORG extracts the operation order number and the designation of the issuing unit. This structure is introduced with characters MSGID/ORDER/. Slashes separate its data fields and double slashes terminate the structure.

This rule provides the first example of lexical cues, which are specified with regular expressions. The DMG recognizes a special syntax for regular expressions that is not described in Kipps (1991). In particular, a sequence of characters delimited by double quotes ("···") is expanded into a regular expression, delimited by parentheses, according to the rules outlined below.

1. Sequences of alphabet characters, digits, and underscores (_) are surrounded by parentheses.
2. Carets (^) and question marks (?) are left as read, as are a backslash (\) and the character immediately following it.
3. Blanks are expanded into a regular expression that matches a sequence of one or more blanks or tabs ([\t]+).
4. Curly braces ({···}) become parentheses. The characters between the curly braces are read as regular expressions.
5. Angle brackets (<···>) and the characters they delimit are left as read.
6. Single quotes ('···') and the characters they delimit are left as read.
7. All other characters (*c*) are expanded into single-character strings ('*c*').

Given these rules, the form

```
"^MSGID%~/%~ORDER%~/<>/<>/<>//"
```

expands into the regular expressions

```
(^ (MSGID) %~ '/' %~ (ORDER) %~ '/') <> ('/') <> ('/') <> ('//')
```

The caret (^) matches the concept "start of line." When random text and nonterminal elements are contained in the string, the string is actually converted into a sequence of regular expressions separated by the random text and nonterminal elements, as seen here. This special syntax was added to improve the readability of regular expressions for matching lexical cues.

The `msg_id` rule also provides the first example of an extractor action. The action

```
{ printf("0/"); prtmap($4); printf("/"); prtmap($2); }
```

outputs a `0/`, followed by the text associated with the structure's fourth element (corresponding to the *dollar-sign variable* `$4`), followed by another slash, and the text of the second element (`$2`). When the OPORD text includes

```
MSGID/ORDER/699TH INF DIV (M)/49003/JUN//
```

the extractor action outputs

```
0/49003/699TH INF DIV (M).
```

The structure element `unitloc` is similarly defined. The function `prtmap()` is defined in the code section of the grammar file.

As seen above, dollar-sign variables, such as `$2` and `$4`, are used in extractor actions to access the *text map* associated with particular elements of a recognized structure rule. In particular, the variable $\$i$ provides access to the text map of the ith element of an n element structure rule ($1 \leq i \leq n$). A double dollar sign (`$$`) is a variable that provides access to the text map of the rule itself. Dollar-sign variables are pointers to C structures of type `zztmap`, the definition and significant fields of which are described in Fig. 6.

```
typedef struct zztextmap ZZTMAP;
struct zztextmap
  {
    int     lineno;   /* line no where text begins */
    int     colmno;   /* no of chars from last newline to text */
    int     nlpos;    /* char position of last newline */
    int     stpos;    /* position of first char of text */
    int     ndpos;    /* position of last char of text */
    char    *lnptr;   /* ptr to start of first line of text */
    char    *nxptr;   /* ptr to first char of text */
    char    *ndptr;   /* ptr to first char following text */
    ZZTMAP  *subt;    /* ptr to subordinate tmaps */
    YYSTYPE value;    /* result of yyparse() */
  }
```

Fig. 6—Text Map C Structure

The structure rule

```
msg_ref_no: "^REF%~/%~{[A-Z]}%~/<>/<date_time_group>//"
   { opdtg = $4->value.dtg; }
```

defines the message reference number of an OPORD, from which OPORG extracts the DTG on which the order is effective. This rule illustrates a slightly more complex action that uses a nonterminal element. The form

```
"^REF%~/%~{[A-Z]}%~/<>/<date_time_group>//"
```

expands into a sequence of five elements

```
(^ (REF) %~ (/) %~ [A-Z] %~ '/') <> ('/')
        <date_time_group> ('//')
```

consisting of a regular expression, followed by a random text element, followed by another regular expression, followed by a nonterminal element, and terminated with another regular expression. This rule matches text such as

```
REF/A/OPLAN 90-10/699TH INF DIV (M)/251200 MAY 90//
```

where the text associated with the nonterminal element is

```
251200 MAY 90.
```

When the extractor encounters the text map for this element, it calls the parser function `yyparse()` requesting that it parse the element's text as an instance of the nonterminal `date_time_group`, which is defined in the grammar file for the parser and is discussed in Section 5. The result of the parse will be assigned to the `value` field of the text map. The type of this field is `YYSTYPE`,

which is also defined in the parser grammar file. For OPORG, YYSTYPE is defined as the union

```
typedef union
  {
    struct dtgroup *dtg;
    int val;
  } YYSTYPE;
```

where dtgroup is a structure representing a DTG. When the extractor action

```
{ opdtg = $4->value.dtg; }
```

is executed, it assigns the dtgroup structure created by parsing

```
251200 MAY 90
```

to the global variable opdtg, to be used later.

The structure rule

```
zone_used: "^TIME ZONE<time_zone>//"
   { printf("/"); prdtg(opdtg); printf("//\n"); }
```

extracts the time zone used throughout the OPORD, again using a nonterminal element. As defined in the grammar file for the parser, the action of the parser on recognizing the nonterminal time_zone is to assign an integer code for the identified zone to the global variable deftimezone. This code is assigned to DTGs for which no time zone is otherwise specified. The extractor action completes the first line of formatted output by printing the effective DTG, which now reflects the appropriate time zone. The function prdtg() is defined in the parser grammar file.

The task organization portion of the heading is defined with the rule

```
task_org: "^HEADING%~/%~TASK ORGANIZATION//"
          <>
          "^/UNITDES /%~UNITLOC /%~CMNTS"
          task_entry+.
```

It is introduced by a heading, possibly separated by some random text from a secondary heading. Note that the headings can also be written

```
"^HEADING%~/%~TASK ORGANIZATION//<>^/UNITDES /%~UNITLOC /%~CMNTS"
```

but, to improve readability, they are not written this way. The body of the task organization section consists of one or more task_entry elements, defined by the rule

```
task_entry: unitdes unitloc cmnts
```

as consisting of the sequence of structure elements unitdes, unitloc, and cmnts.

The structure rules for unitdes, unitloc, and cmnts provide additional examples of extraction actions. The rule

```
unitdes: (\n? '/'?) (%P+ (' ' %P+)*) ([ \t]+)
   { printf("1/"); prtmap($2); }
```

defines unitdes as a sequence of printing characters (signified by the built-in lexical macro %P) with single blanks interspersed, beginning on a new line and optionally prefixed by a slash, e.g.,

```
/TF 2-74 INF, 3D BDE, 21ST ID(L).
```

Question marks (?) denote optionality. The three regular expressions of the rule are kept distinct so that the characters actually making up the unit designation can be accessed independently of the characters delimiting it. In particular, when the extractor action is executed, it outputs a 1/, followed by the text associated with the second regular expression, as in

```
1/TF 2-74 INF, 3D BDE, 21ST ID(L).
```

The structure element unitloc is similarly defined.

The structure rules for cmnts

```
cmnts: ('/'?) cmnttxt ('//'|'.')
   { printf("2/"); prtmap($2); printf("//\n"); }
```

and for cmnttxt

```
cmnttxt: "{(OPC|ATT)[A-Z]*} TO <> EF<date_time_group>"
   { printf("/"); prtmap($2);
     printf("/"); prdtg($4->value.dtg);
     printf("//\n"); }
```

illustrate the order of action execution. These rules will match text such as

```
/OPCON TO 1ST BDE EFFECTIVE 291300 MAY 90.
```

where the text associated with the nonterminal element <date_time_group> is

```
FECTIVE 291300 MAY 90.
```

When the extractor encounters the text map for cmnts, it finds an extractor action. Before it executes this action, the extractor first traverses the subordinate text maps, where it encounters the text map for cmnttxt, which also has an extractor action, as well as more subordinate text maps. Of these, the extractor encounters the nonterminal element <date_time_group> and calls the parser on this element's text. As the extractor cannot descend further down the text map hierarchy, it returns to the cmnttxt text map and executes its extractor action, which outputs the remaining data of the task organization entry. Only then does the extractor return to the cmnts text map and execute its extractor action, which

outputs the next line of data, which is a comment containing the entire text of `cmnttxt`.

The Extractor Code

The second panel of Fig. 5 contains the code section of the example grammar file and defines two functions, `prtmap()` and `main()`. The function `prtmap()` outputs the text associated with a given text map. The predefined function `zzfilltmap()` is called to fill the character pointer fields of the text map, `lnptr`, `nxptr`, and `ndptr`, from a global character array that contains all the characters from the input document.

The function `main()` is the top-level function for the resulting translation system. It first calls the predefined mapper function `zzmap()`, which, if successful, will return the text map for the root element, i.e., `opord`. The extractor function `zzextract()`, also predefined, is called on the root text map and does a top-down traversal of the map, looking for nonterminal elements and structure elements with associated extractor actions as outlined above. When `zzextract()` encounters a nonterminal element, it calls the parser function `yyparse()` to parse the text of the element as an instance of that nonterminal, which must be defined as one of the *start symbols* of the parser. The result of the parser is assigned to the `value` field of the element's text map. When `zzextract()` encounters the text map of a structure element, it calls itself recursively on the subordinate text maps before executing any associated extractor action.

The function `main()` also contains a call to `zzwrttmap()` in a comment. If necessary, this call can be uncommented to examine the complete text map hierarchy of an input OPORD. Each text map in the hierarchy will be output using the general form

ddd: *name*(*action*) *l*(*c*) *from*–*to*

where *ddd* is the text map number, *name* is the name of the element, (*action*) is the action number (see the function `zztmapact()` in `y.map.c`), *l* is the line number on which the text of the element begins, *c* is the column on that line on which the text begins, *from* is the character position in the file on which the text begins, and *to* is the character position on which the text ends, exclusive.

5. PARSING DATE-TIME GROUPS WITH RACK

The RAND Compiler Kit (RACK) constructs the parser program accessed by the extractor. RACK was built as part of an in-house effort to make parsing technology developed under the DARPA-sponsored ROSIE Language project (Kipps, 1988) readily available to others at RAND and elsewhere. An overview of the RACK system can be found in Kipps (1990); complete documentation can be found in Kipps (1991). In this section, we briefly overview RACK and describe its application to the task of parsing DTGs.

THE RAND COMPILER KIT: RACK

RACK is a parser generator for advanced applications. RACK parsers are unique in their ability to recognize non-LR(k) languages. This means that they can look ahead an arbitrary number of symbols to determine correct grammatical structure. Conventional parser generators, such as YACC (Johnson, 1975), produce parsers that have at most one symbol of look-ahead.

RACK generates practical parsers using Tomita's algorithm (Tomita, 1985) for fast general context-free parsing. The execution speed and memory requirements of RACK parsers compare favorably with parsers generated by YACC. RACK is fully upward compatible with YACC; programmers familiar with YACC should have little difficulty learning to use RACK's additional features. RACK is implemented in portable C and can generate both C and Lisp translators.

The input to RACK is a grammar file that describes the lexical elements and the syntax of a target language, such as the language of DTGs. The output is a parser program file (or parser) and a scanner program file (or scanner) for parsing input text of the target language. The user specifies the programming language in which the parser and scanner are to be coded, called the implementation language. Presently, RACK supports three implementation languages: C, Common Lisp, and Standard Lisp. The OPORG system uses a C-based parser.

The relationship of RACK to the parser and scanner is depicted in Fig. 7. The parser and scanner are intended to operate as a front-end to a user program, such as the extractor. The user program invokes the parser with the function `yyparse()` when it needs to translate textual input into a representation that it

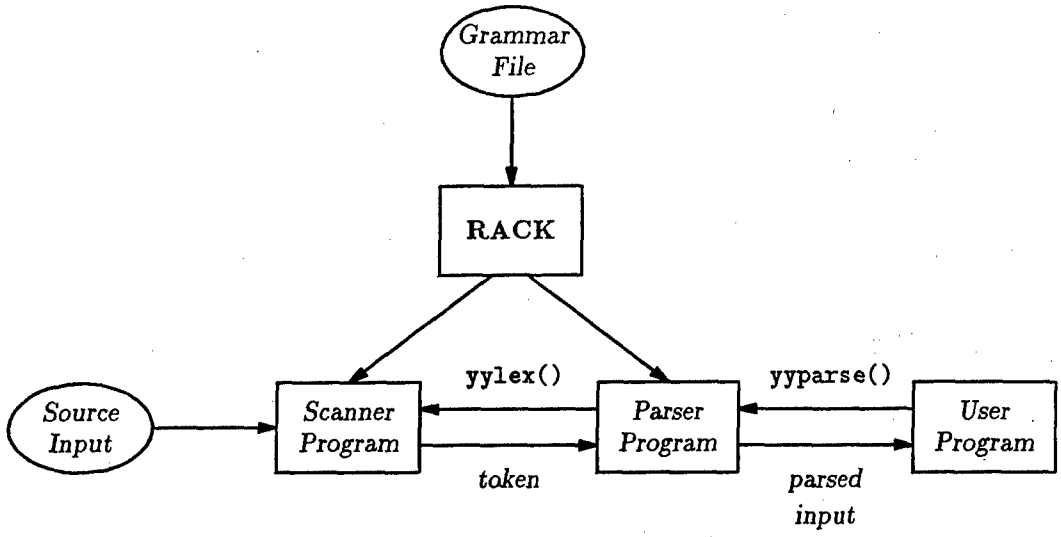

Fig. 7—RACK System Architecture

can easily access and use. The parser repeatedly requests tokens from the scanner, which it invokes with the function `yylex()`, until it recognizes a sentence in the target language or encounters a syntax error.

RACK GRAMMAR FILES

A RACK grammar file has the basic format:

Parser Declarations
%lex
Scanner Declarations
%%
Lexical Rules
%%
Grammar Rules
%%
Code.

Parser declarations are used to define the tokens and start symbols of the target language. Start symbols are distinguished nonterminals denoting root syntactic constructs. Declarations are also used to specify additional code segments, such as `#include` statements, that should appear at the front of the parser program file. Scanner declarations are used to specify which characters denote escape and end-of-sentence and which begin and end comments. They are also used to define the class of printing and white space characters, to specify case dependencies, and to enumerate the keywords of a language. The *lexical rules* section consists of one or more *lexical rules*, which describe character patterns for recognizing the tokens of the target language. The *grammar rules* section consists of one or more

grammar rules, which describe the syntax of nonterminals in the target language. The *code* section can be used to define additional subroutines to be added to the parser program. Each section is separated by double percent signs (%%). The scanner declarations are separated from the parser declarations by the keyword `%lex`.

Lexical rules have the general form

[*token* :] *pattern* [*action*] [;]

where square brackets ([···]) denote optional constructs. When present, *token* may be a token name, such as `IDENT`, a token number, such as 257, or a character literal, such as `'='`. On recognizing *pattern*, the scanner executes *action* and returns the token number associated with *token*. If *token* is missing and *action* is present, *action* is expected to return a token number, which is returned by the scanner. If *action* executes the internal function `YYRESET`, then the matched characters are returned to the input stream and the scanner restarts. If *action* executes the internal function `YYRESTART`, then the matched characters are discarded and the scanner restarts from the current state of the input stream. If both *token* and *action* are missing, the default action is to execute `YYRESTART`. The characters matched by the rule are assigned to the global variable `yytext`, a string of length `yyleng`. These characters can be accessed by *action*.

The *pattern* portion of a lexical rule is a sequence of three *regular expressions* (character patterns) separated by a slash (/); for a discussion of regular expressions and their use in recognizing character strings, see Aho and Ullman (1977) for a discussion of regular expressions and their use in RACK, and Kipps (1991). When all three regular expressions are included, *pattern* has the form

[*lc*] / *re* / [*rc*].

The regular expressions *lc* and *rc* are respectively called the *left* and *right context* of the pattern; *lc* and *rc* are both optional. The regular expression *re* is called the *primary* of the pattern and must be present. When both *lc* and *rc* are excluded, the slashes (/) can be discarded.

The scanner matches input characters against the patterns of the lexical rules and selects the rule whose pattern completely matches the longest sequence of characters. In a tie, the rule appearing earliest in the grammar file is selected. Once selected, the input characters that match the left context *lc* are discarded; those matching the primary *re* are saved in the global variable `yytext`; those matching the right context *rc* are returned to the input buffer.

Grammar rules have the general form

name : [*name*|*literal*|*action*]* [%prec [*name*|*literal*]] [*action*] [;]

where square brackets ([···]) again denote optional constructs, bars (|) denote disjunction, and stars (*) denote repetition. The left-hand side of a rule, *name*, is the name of a nonterminal. The right-hand side consists of an arbitrary sequence of token and nonterminal names, character literals, and actions. When the parser recognizes an instance of a grammar rule in the input text, it executes the rule's terminating *action*. Actions may also appear anywhere in the right-hand side of the rule. If the parser is not following multiple derivations, then these actions are executed during the recognition of the rule; otherwise, action execution is delayed, including execution of terminating actions. Further discussion on multitrack parsing and delayed actions can be found in Kipps (1991).

As with extractor actions, communication between grammar rule actions is supported with the use of dollar-sign variables. To return a value, a grammar rule action assigns that value to the variable `$$`. To read a value returned by an earlier action, a rule action examines the variables `$1`, `$2`, `$3`, etc. These variables refer to values associated with (or returned by) the components of the right-hand side of the rule, as read from left to right. The value associated with a token must be assigned to the variable `yylval` by the action of a lexical rule. The type of dollar-sign variables is `YYSTYPE`, which defaults to `int`. This can be changed with the `%union` parser declaration.

THE DTG SYNTACTIC GRAMMAR

The syntactic grammar for parsing time zones and DTGs used in the OPORG system is shown in Fig. 8 as three panels. In the remainder of this section, we will describe the meaning and use of various pieces of this grammar.

The Declarations Section

The first panel of Fig. 8 contains the parser and scanner declarations. The first parser declaration

```
%coding C Language
```

specifies that the implementation language for the parser and scanner is intended to be C.

```
%coding C Language
{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
typedef struct dtgroup
  {
    int day;
    int timefm;
    int timeto;
    int zone;
    int month;
    int year;
  } DTG;
}
%union
  {
    struct dtgroup *dtg;
    int val;
  }
%token <val> DAY MONTH TIME ZONE YEAR
%type  <val> time_zone
%type  <dtg> date_time_group dtg
%start date_time_group time_zone
%lex
{
#include <stdlib.h>
int toint(s)
  char *s;
  { int v = 0;
    do { v = 10*v + (*s - '0'); }
      while(*(++s) != '\0');
    return v;
  }
}
%readtmaps
%input %upcase
%def %DAY   ([0-9]<2>)
%def %TIME  ([0-9]<4>)
%def %ZONE  ([A-Z])
%def %MONTH ((JAN|FEB|MAR|APR|MAY|JUN|JUL|AUG|SEP|OCT|NOV|DEC) [A-Z]*)
%def %YEAR  ((19)? [0-9]<2>)
```

Fig. 8—Example DTG Syntactic Grammar (Declarations)

```
%%
%dtg
(%DAY %TIME '-' %TIME %W* %ZONE %W* %MONTH %W* %YEAR)
    { %dtg-; %day+; %timefm+; %timeto+; %zone+; %month+; YYRESET; }
(%DAY %TIME '-' %TIME %W* %MONTH %W* %YEAR)
    { %dtg-; %day+; %timefm+; %timeto+; %month+; YYRESET; }
(%DAY %TIME %W* %ZONE %W* %MONTH %W* %YEAR)
    { %dtg-; %day+; %timefm+; %zone+; %month+; YYRESET; }
(%DAY %TIME %W* %MONTH %W* %YEAR)
    { %dtg-; %day+; %timefm+; %month+; YYRESET; }
.   { YYRESTART; }
%day
DAY:   / %DAY / [0-9]       { %day-; yylval.val = toint(yytext); }
%timefm
TIME:  / %TIME / [^0-9]     { %timefm-; yylval.val = toint(yytext); }
%timeto
TIME:  '-' / %TIME /        { %timeto-; yylval.val = toint(yytext); }
%zone
ZONE:  %ZONE / %W* %MONTH   { %zone-; yylval.val = *yytext; }
ZONE:  '/' / %ZONE /        { %zone-; yylval.val = *yytext; }
%month
MONTH: JAN [A-Z]*           { %month-; %year+; yylval.val =  1; } |
       FEB [A-Z]*           { %month-; %year+; yylval.val =  2; } |
       MAR [A-Z]*           { %month-; %year+; yylval.val =  3; } |
       APR [A-Z]*           { %month-; %year+; yylval.val =  4; } |
       MAY [A-Z]*           { %month-; %year+; yylval.val =  5; } |
       JUN [A-Z]*           { %month-; %year+; yylval.val =  6; } |
       JUL [A-Z]*           { %month-; %year+; yylval.val =  7; } |
       AUG [A-Z]*           { %month-; %year+; yylval.val =  8; } |
       SEP [A-Z]*           { %month-; %year+; yylval.val =  9; } |
       OCT [A-Z]*           { %month-; %year+; yylval.val = 10; } |
       NOV [A-Z]*           { %month-; %year+; yylval.val = 11; } |
       DEC [A-Z]*           { %month-; %year+; yylval.val = 12; }
%year
YEAR:  (19)? / [0-9]<2> / { %year-; yylval.val = toint(yytext); }
```

Fig. 8 (cont.)—Example DTG Syntactic Grammar (Lexical Rules)

```
%%
time_zone: %zone+ ZONE          { deftimezone = $2; };

date_time_group: %dtg+ dtg      { $$ = $2; };

dtg: /* empty */                { $$ = NULL; }                          |
     DAY TIME MONTH YEAR        { $$ = mkdtg($1,$2,0,$3,$4,'\0'); }     |
     DAY TIME ZONE MONTH YEAR   { $$ = mkdtg($1,$2,0,$4,$5,$3); }       |
     DAY TIME TIME MONTH YEAR   { $$ = mkdtg($1,$2,$3,$4,$5,'\0'); }    |
     DAY TIME TIME ZONE MONTH YEAR
                                { $$ = mkdtg($1,$2,$3,$5,$6,$4); };

%%
yyerror(s)
  char *s;
  { printf("%s: line no = %d\n", s, yylineno); }

int deftimezone = ' ';

DTG *mkdtg(day, timefm, timeto, month, year, zone)
  int day, timefm, timeto, month, year, zone;
  { DTG *dtg;
    dtg = (DTG*)calloc(1, sizeof(DTG));
    dtg->day    = day;
    dtg->timefm = timefm;
    dtg->timeto = timeto;
    dtg->zone   = zone;
    dtg->month  = month;
    dtg->year   = year;
    return(dtg);
  }

prdtg(dtg)
  DTG *dtg;
  {
    if(!dtg) return;
    if(dtg->zone == '\0') dtg->zone = deftimezone;
    printf("%02d%04d%c%02d%02d", dtg->day, dtg->timefm,
            dtg->zone, dtg->month, dtg->year);
  }
```

Fig. 8 (cont.)—Example DTG Syntactic Grammar (Grammar Rules and Code)

Appearing next is a segment of user code.

```
{
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
typedef struct dtgroup
  {
    int day;
    int timefm;
    int timeto;
    int zone;
    int month;
    int year;
  } DTG;
}
```

This code segment includes three standard C library files and defines a data structure for representing a DTG. Characters between the delimiting curly braces will be inserted at the beginning of the parser program file.

The next declaration

```
%union
  {
    struct dtgroup *dtg;
    int val;
  }
```

defines the type of elements on the parser's value stack as either a pointer to a `dtgroup` (called `dtg`) or `int` (called `val`). In the parser and scanner program files, the type YYSTYPE will be defined as this union.

The declarations

```
%token <val> DAY MONTH TIME ZONE YEAR
%type  <val> time_zone
%type  <dtg> date_time_group dtg
```

name the tokens of the grammar and assign types to the values associated with its tokens and nonterminals. In particular, all tokens and the nonterminal `time_zone` have associated values of type `int`, while the nonterminals `date_time_group` and `dtg` have associated values of type `struct dtgroup`.

The final parser declaration

```
%start date_time_group time_zone
```

distinguishes the nonterminals `date_time_group` and `time_zone` as start symbols of the grammar.

The scanner declarations begin with a segment of user code to build integers from RACK strings

```
{
#include <stdlib.h>
int toint(s)
  char *s;
  { int v = 0;
    do { v = 10*v + (*s - '0'); }
      while(*(++s) != '\0');
    return v;
  }
}
```

This function will be inserted at the beginning of the scanner program file. This code includes a standard C library file and defines a function `toint()` that converts a string of digits to an integer.

The declaration

```
%readtmaps
```

tells the scanner to expect to read input from text maps supplied by the extractor, while the declaration

```
%input %upcase
```

specifies that all input alphabet characters should be converted to uppercase when read.

The declarations

```
%def %DAY   ([0-9]<2>)
%def %TIME  ([0-9]<4>)
%def %ZONE  ([A-Z])
%def %MONTH ((JAN|FEB|MAR|APR| ··· |NOV|DEC) [A-Z]*)
%def %YEAR  ((19)? [0-9]<2>)
```

define a set of lexical macros, which have the general form

```
%def macro (re)
```

This form assigns the regular expression *re* to the lexical macro *macro*, a symbol or nonalphabet character prefixed by a percent sign. When *macro* is encountered in the body of a lexical rule, it is replaced by *re*. The regular expression assigned to %DAY matches two consecutive digits, while that assigned to %TIME matches four digits, and that of %ZONE matches a single letter. The regular expression assigned to %MONTH matches any of the first three letters in the name of a month, as well as any alphabet characters immediately following those letters. Finally, the regular expression assigned to %YEAR matches the four digits in a year or, optionally, the last two digits of a year.

The Lexical Rules

The second panel of Fig. 8 defines the lexical rules of the grammar in seven *lexical states*. A lexical state encapsulates a set of lexical rules, which can be enabled and disabled from lexical or grammar rule actions. A lexical state is defined with a symbol (prefixed by a percent sign) appearing before a set of lexical rules. All rules up to the next lexical state definition (or the end of the lexical rule section) belong to the named state. A lexical state's rules are enabled by suffixing it with a plus (+) and disabled with a minus (-).

The first lexical state, %dtg, discards input characters until a DTG is at the beginning of the input buffer. In general, a DTG is defined as a six-digit number expressing date and time. The first two digits indicate the day of the month and the last four digits indicate the time. The time zone, month, and year are added to avoid confusion. In addition, the time can be followed by a dash (-) and four more digits, indicating a time range. A complete DTG would appear as

```
241000-1300Z JANUARY 1990.
```

The grammar seen in Fig. 8 recognizes four different DTG forms: with and without a time-to part, and with and without a zone. The grammar actually used by the OPORG system recognizes two additional cases: without a time-from and time-to part, and without a time-to, zone, month, and year. These two cases are straightforward additions to the grammar. They were excluded from Fig. 8 to simplify the description of the grammar.

The first four rules of the lexical state %dtg

```
(%DAY %TIME '-' %TIME %W* %ZONE %W* %MONTH %W* %YEAR)
   { %dtg-; %day+; %timefm+; %timeto+; %zone+;
     %month+; YYRESET; }
(%DAY %TIME '-' %TIME %W* %MONTH %W* %YEAR)
   { %dtg-; %day+; %timefm+; %timeto+; %month+; YYRESET; }
(%DAY %TIME %W* %ZONE %W* %MONTH %W* %YEAR)
   { %dtg-; %day+; %timefm+; %zone+; %month+; YYRESET; }
(%DAY %TIME %W* %MONTH %W* %YEAR)
   { %dtg-; %day+; %timefm+; %month+; YYRESET; }
```

specify regular expressions that match the four different DTG forms. When one of these forms is recognized, the rule's action disables the state %dtg and enables the lexical state used to match the individual parts appearing in the DTG. The call to YYRESET returns the matched characters to the input buffer and restarts the scanner with the new lexical states enabled. This ensures that a DTG is at the front of the input buffer.

The fifth rule of %dtg

```
. { YYRESTART; }
```

successively removes characters from the input buffer when a DTG is not present. The dot (.) is a regular expression that matches any character except newline and end-of-sentence. The call to YYRESTART restarts the scanner from the current position of the input buffer. Because newline is a white space character, it will automatically be discarded by the scanner. Because the scanner prefers matches that account for the longest input sequence, there is no ambiguity in these rules.

The next three lexical states contain one rule each. The lexical rule

```
DAY: / %DAY / [0-9]
   { %day-; yylval.val = toint(yytext); }
```

in state %day recognizes two digits as the day portion of a DTG if they are followed by another digit. The action of this rule disables the state and assigns the integer value of the two digits to yylval.val. The variable yytext is a null-terminated string containing the characters matched by the primary regular expression of a lexical rule's pattern. The variable yylval is of type YYSTYPE and is used to pass values from the scanner to the grammar rule actions of the parser. The function toint() is defined in the scanner declarations. The scanner returns the token number associated with DAY to the parser, indicating which token was recognized.

The lexical rule

```
TIME: / %TIME / [^0-9]
   { %timefm-; yylval.val = toint(yytext); }
```

recognizes the time portion of the DTG. The regular expression [^0-9] matches any nondigit and keeps the scanner from mistaking the first four digits of a DTG as the time. The lexical rule

```
TIME: '-' / %TIME /
   { %timeto-; yylval.val = toint(yytext); }
```

recognizes the time-to portion of a DTG, if present.

The next lexical state, %zone, contains the two rules

```
ZONE: %ZONE / %W* %MONTH
   { %zone-; yylval.val = *yytext; }

ZONE: '/' / %ZONE /
   { %zone-; yylval.val = *yytext; }
```

matching a time zone designation. The first rule matches a time zone appearing in a DTG; the second rule matches a time zone from the time zone designation portion of an OPORD's heading. The time zone is expected to be a single alphabet character, and its integer character code is assigned to yylval.val.

The lexical state %month contains 12 alternatives of the same token. The rule

```
MONTH: JAN [A-Z]*  { %month-; %year+; yylval.val =  1; } |
       FEB [A-Z]*  { %month-; %year+; yylval.val =  2; } |
       MAR [A-Z]*  { %month-; %year+; yylval.val =  3; } |
       APR [A-Z]*  { %month-; %year+; yylval.val =  4; } |
       MAY [A-Z]*  { %month-; %year+; yylval.val =  5; } |
       JUN [A-Z]*  { %month-; %year+; yylval.val =  6; } |
       JUL [A-Z]*  { %month-; %year+; yylval.val =  7; } |
       AUG [A-Z]*  { %month-; %year+; yylval.val =  8; } |
       SEP [A-Z]*  { %month-; %year+; yylval.val =  9; } |
       OCT [A-Z]*  { %month-; %year+; yylval.val = 10; } |
       NOV [A-Z]*  { %month-; %year+; yylval.val = 11; } |
       DEC [A-Z]*  { %month-; %year+; yylval.val = 12; }
```

assigns the integer code for the recognized month to `yylval.val` and causes the scanner to return the token number associated with the token `MONTH`. This rule also enables the lexical state %year, which was not previously enabled because it would conflict with the rule in state %day. The rule in state %year

```
YEAR: (19)? / [0-9]<2> /
   { %year-; yylval.val = toint(yytext); }
```

assigns the integer value of the last two digits of the year to `yylval.val`.

The Grammar Rules

The third panel of Fig. 8 defines three grammar rules and additional user code. The first rule

```
time_zone: %zone+ ZONE
   { deftimezone = $2; };
```

defines the syntax of the start symbol `time_zone`. The form %zone+ is a rule action that enables the lexical state %zone, which allows the scanner to recognize the token `ZONE` in the input text. The character code of the zone, which is assigned to `yylval.val`, is accessed in the grammar rule action

```
{ deftimezone = $2; }
```

with the dollar-sign variable $2 and assigned to the global variable `deftimezone`.

The next rule

```
date_time_group: %dtg+ dtg  { $$ = $2; };
```

defines the syntax of the other start symbol `date_time_group`. It enables the %dtg lexical state with rule action %dtg+ and defines the start symbol as an instance of the nonterminal `dtg`. By assigning the value of $2 to $$, the rule action associates the value of `dtg` with `date_time_group`, which will be passed out of the parser to the extractor.

Finally, the rule

```
dtg: /* empty */  { $$ = NULL; }                          |
     DAY TIME MONTH YEAR
                  { $$ = mkdtg($1,$2,0,$3,$4,'\0'); }    |
     DAY TIME ZONE MONTH YEAR
                  { $$ = mkdtg($1,$2,0,$4,$5,$3); }      |
     DAY TIME TIME MONTH YEAR
                  { $$ = mkdtg($1,$2,$3,$4,$5,'\0'); }   |
     DAY TIME TIME ZONE MONTH YEAR
                  { $$ = mkdtg($1,$2,$3,$5,$6,$4); };
```

defines four alternatives of the nonterminal dtg, corresponding to the four DTG patterns defined in the lexical state %dtg. This rule also defines a fifth alternative, that there is no DTG mentioned in the input text.

The code section of the grammar defines three functions and a global variable. The function yyerror() is required by the parser and is called whenever the parser encounters a syntax error; more sophisticated error handling can be implemented (Kipps, 1991). The global variable deftimezone will be assigned the character code of the time zone used throughout the order. By default, it is assigned a blank space. The function mkdtg() allocates and initializes a dtgroup data structure, while the function prdtg() outputs the data in a dtgroup structure, assigning the default zone if the time zone is not already defined; the timeto portion of a DTG is ignored.

6. THE OPORG TRANSLATOR SYSTEM

In this section, we describe how to install the OPORG system on an IBM PC under MS-DOS.[1] The individual source files of the OPORG translator system must first be generated under UNIX and then transferred to the target PC environment. We assume that the OPORD structural grammar (see Fig. 5) is contained in a text file called `tskorg.y` and that the DTG syntactic grammar (see Fig. 8) is contained in text file `dtg.y`. We also assume that the target PC environment has a C compiler, such as Turbo C (TurboC, 1988).[2] We use a percent sign (%) to indicate the UNIX prompt and a right angle bracket (>) to indicate the MS-DOS prompt.

Under UNIX, the mapper and extractor programs are generated by calling the DMG on the structural grammar with the **dmg** command, i.e.,

```
% dmg tskorg.y.
```

DMG creates the file `y.map.c`, which contains the source code for the mapper and extractor.

The parser and scanner programs are generated by calling RACK on the syntactic grammar with the **yacc** command, i.e.,

```
% yacc -d dtg.y.
```

RACK subsumes the UNIX command **yacc** in order to take advantage of YACC-specific features already existing in UNIX tools, such as **make**. This call creates three files: `y.lex.c`, which contains the source code for the scanner; `y.tab.c`, which contains the source code for the parser; and `y.tab.h`, which contains parser declarations that are required by the extractor.

The unusual looking names of the mapper, parser, and scanner files conform to YACC's naming convention, but they may not transfer correctly to a PC environment. For this reason, these files are expected to be renamed to eliminate the first dot (.). This can be done with the UNIX **mv** command, i.e.,

```
% mv y.map.c ymap.c
% mv y.tab.c ytab.c
% mv y.tab.h ytab.h
% mv y.lex.c ylex.c.
```

Then transfer these files to the PC file system. There are several readily available file transfer mechanisms, such as the public domain Kermit program and UUCP.

[1] MS-DOS is a registered trademark of Microsoft Corporation.

[2] Turbo C is a registered trademark of Borland International.

Once on the PC, the OPORG system source files must be compiled and linked. Assuming the C compiler is called cc, this can be done with the call

```
> cc -o oporg ylex.c ytab.c ymap.c.
```

This call creates the executable file oporg, which constitutes the OPORG translator system command. Figure 9 shows the contents of a file for constructing the system using the Turbo make command.

The oporg command has the form

oporg *file*

where *file* is the name of the OPORD text file. OPORG extracts the relevant task organization data and writes this data to standard output, which can be redirected to a text file. OPORG writes the extracted data according to the format outlined below.

0/opord no/issuing unit/DTG order effective//
1/unit des/unit loc/attached to/DTG effective//
2/comment//
3/DTG operation to begin//

The first character of each line of output identifies the fields of data on the line. A zero (0) indicates that the line contains the operation order number, the issuing unit designation, and DTG the OPORD is effective. A one (1) indicates that the line contains data from the task organization section of the OPORD, namely, a unit designation and location, the unit to which it is being attached or OPCONed, and DTG effective (if given). A two (2) indicates a comment, which a computer system should ignore; comments are used to highlight portions of OPORD text from which data is extracted for manual verification. A three (3) indicates that the line contains the DTG for the beginning of the operation. Data fields are separated by a slash (/); data lines are terminated by a double slash (//).

For example, if the portion of text seen earlier in Fig. 1 is contained in the text file opord1, then the call

```
% oporg opord1
```

writes

```
0/49003/699TH INF DIV (M)/251200U0590//
1/TF 2-13 MECH/NL1702/699TH AB AS DIV TCF/021200U0690//
2/OPCON TO 699TH AB AS DIV TCF EFFECTIVE 021200 JUN 90//
1/TF 1-61 AR/NL6392/3D BDE (LIVE FIRE)///
2/OPCON TO 3D BDE (LIVE FIRE) EFFECTIVE ON CLOSURE IN 3D BDE M//
1/TF 3-4 AVN/AREA B/1ST BDE/291300U0590//
2/OPCON TO 1ST BDE EFFECTIVE 291300 MAY 90//
3/020001U0690//
```

to standard output.

```
# Makefile
#
# for compiling OPORG source files on IBM PC compatible computer.

all:    ytab.obj ylex.obj ymap.obj
        tcc -mh -DZZCBUFSIZE=30000 -w-pia -N -eopmap \
        ylex.obj ytab.obj ymap.obj

ytab.obj:       ytab.h ytab.c
        tcc -mh -DZZCBUFSIZE=30000 -w-pia -N  -c ytab.c

ylex.obj:       ylex.c
        tcc -mh -DZZCBUFSIZE=30000 -w-pia -N  -c ylex.c

ymap.obj:       ytab.h ymap.c
        tcc -mh -DZZCBUFSIZE=30000 -w-pia -N -c ymap.c
```

Fig. 9—Makefile for Constructing OPORG

7. CONCLUSIONS

The UATOP project has investigated the problems associated with extracting tactical data from combat operation orders (OPORDs). Although OPORDs contain a level of tactical information appropriate for many military software systems, such as those for battle management, logistics, war-gaming, and combat training, this information is currently inaccessible as computer data, except by manual extraction and entry. Using tools developed by the UATOP project, we have demonstrated that concise and clear rules can be used to automate the extraction of OPORD data.

We have demonstrated an application of our approach by describing the structure and operation of the OPORG translation system. OPORG extracts task organization data from input OPORDs and outputs this data using a structured format that can readily be accessed by other software systems. The OPORG system illustrates that our approach to OPORD translation is flexible, reliable, and portable. The OPORG system is generated automatically from grammar files by two translator-generator tools: DMG and RACK. Modifications can be added to the system easily by editing the grammar files. Although the OPORG system files must be generated under the UNIX operating system, once generated, OPORG can be run on an IBM PC-compatible computer. The OPORG system source files are written entirely in portable C.

We have also applied this approach to the larger task of translating OPORDs into simulation scenarios for use by the ACAC simulation system. In this task, the data being extracted are the directives to maneuver units, which are output as *scripts* that drive the simulation model. A preliminary translation system, called OPSCRIPT, has been implemented and is demonstrable. The OPSCRIPT translation system will be described in a later document.

Given its generic nature, we feel that our approach to extracting data from OPORDs can be applied to other highly structured but "not machine-readable" documents. Our results demonstrate that regularity and structure can transform otherwise untranslatable pieces of text into machine-readable data files. These results recommend that the DoD should strive to standardize the format and language of its documents as much as possible to make them available as input to automated military systems.

REFERENCES

AFSC, *Joint Staff Officer's Guide*, No. AFSC Pub 1, Armed Forces Staff College, 1984.

Aho, A. V., and J. D. Ullman, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1972.

Aho, A. V., and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley Publishing Co., Reading, Mass., 1977.

Johnson, S. C., *YACC–Yet Another Compiler Compiler*, CSTR 32, Bell Laboratories, Murray Hill, N.J., 1975.

Kipps, J. R., *A Table-Driven Approach to Fast Context-Free Parsing*, RAND, N-2841-DARPA, December 1988.

Kipps, J. R., "RACK: A Parser Generator for AI Languages," in *Proceedings of IEEE International Conference on Tools for AI*, November 1990, pp. 430–435.

Kipps, J. R., *The RAND Compiler Kit (RACK): Reference Manual and User's Guide*, RAND, N-3100-RC, 1991.

Tomita, M., "An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications," Ph.D. dissertation, Computer Science Department, Carnegie Mellon University, Pittsburg, Pa., 1985.

TurboC, *Turbo C: Reference Guide Version 2.0*, Borland International, Scotts Valley, Cal., 1988.

U.S. Army, *JANUS-T Simulation System*, unpublished documentation.

U.S. Army, *Operations*, FM 100-5, U.S. Army.

U.S. Army, *SCALP Scenario-determined, Computer Assisted Logistics Planning*, technical report, U.S. Army Logistics Center, September 1989.